Defining .NET Components with Namespaces

NDepend White-Book

v1.0 May 2011

A .NET software component is a compiled set of classes that provide a programmable interface that is used by consumer applications for a service. As a component is no more than a logical grouping of classes, what then is the best way to define the boundaries of a component within the .NET framework? How should the classes inter-operate? The NDepend team discusses the issues and comes up with a solution.

This white-book will attempt to explain the advantages of using namespaces to define the boundaries of components in a .NET assembly, to suggest the best ways of using namespaces as components, and to describe how to continuously enforce the relevant rules of namespace dependency.

The aim is to assist a .NET development shop to rationalize the development of a large code base so as to let the developers become productive, and to reduce the cost of maintenance.

The advice given in this white-book comes from experience gained in years of real-world consulting and development in various development corporations. It has proved to be effective several times, in different circumstances.

In the white-book **Partitioning code base through .NET assemblies and Visual Studio Projects** we focused on assemblies, and the reasons why it is more convenient to have fewer and larger assemblies. We suggested how to organize the VS solutions and VS projects. We mentioned that a component is a finer-grained logical concept than a physical assembly. This implies that a large assembly is not a monolithic piece of code, but is likely to contain several components.

Defining components inside .NET assemblies

The .NET platform has no intrinsic means of defining a component inside an assembly. So what is a component? There are whole books that are dedicated to explaining what a component is but, to keep things simple and practical, we'll list some definitions. A component can be:

- **An aggregate of classes and associated types:** A class is mostly too lightweight to define a component. Therefore a component will almost always include several classes or interfaces and their associated types (enumerations, structures, exceptions, attributes, delegates…). All these types are grouped by a common well-understood semantic concept that gives its name to the component.
- **A unit of development and test:** The notion of a component is well-suited to the way that code is developed. When adding a new feature to the product, a developer generally dedicates time to develop a new set of cohesive classes and their associated tests. This represents a component, or more likely a group of related components. Because of this, the organization of components often mirrors the set of features; but a component can also host some infrastructure classes, such as domain classes or helpers and utility classes.
- **A unit of learning for developers:** It is a daunting task to have to reverse-engineer a large code base. It helps to use components to partition the code into reasonably-sized chunks that can be learned more easily than if they were entangled within a monolithic style code base.
- **A unit of architecture:** One of the most challenging tasks for any real-world code-base development is to master the entropy that necessarily arises in such a complex system. We can do this by imposing an architecture on a code base through a well-defined set of components. This is an adaptation of the '*Divide and Conquer'* principle introduced in 1628 in the *Discourse of Method* of *René Descartes*, inventor of modern science. Relying on well-defined units to *divide and conquer* represents the way that scientists and engineers have worked over the ages to elucidate complex systems.
- **A unit of layering:** In order to rationalize a complex system, It is not enough just to have well-defined components: They must be properly layered. In other words, the graph of dependencies between components must be a *Direct Acyclic Graph* (DAG). If the graph of dependencies between components contains a cycle, components involved in the cycle cannot be developed and tested independently. Because of this, the cycle of components represents a super-component, with higher entropy than the sum of the entropies of its contained components.

For this article, we'll use the concept of a component as being a well-defined aggregate of types, with a reasonable size, and with an acyclic graph of dependencies between components.

Using namespaces to define components

A component is well-defined if its boundaries are explicit. Only two .NET language constructs can be used to explicitly define component boundaries and contain a set of children types: Parent class and namespace.

The advantage of a parent class over a namespace is that nested classes can be declared with a visibility level. This makes it possible to have class encapsulation in the component. On the other hand, the advantage of namespace over parent class is that the namespace is a language artifact and not a CLR artifact. Namespaces are absent from assembly metadata. The namespace name is just a prefix added to contained classes. This means that a namespace is more lightweight than a parent class. Also, when using a parent class, there is a risk of confusion between a parent class and an application class. On the other hand there is no risk of confusing a namespace with a class.

We therefore prefer to use namespaces to define explicit component boundaries. Namespaces are also often used to organize public API presentation. This usage of namespace fits nicely with the concept of components as we've shown: *Component organization often mirrors the set of features, but a component can also host some infrastructure classes*.

Size of components

We measure the size of a component by counting the lines of code. A **logical Line of Code (Loc)** represents a Sequence Point. A sequence point is the code excerpt highlighted in dark red in the VS code editor window, when creating a breakpoint. Most of .NET tools for developers, including VS and NDepend, measure Lines of Code through sequence points.

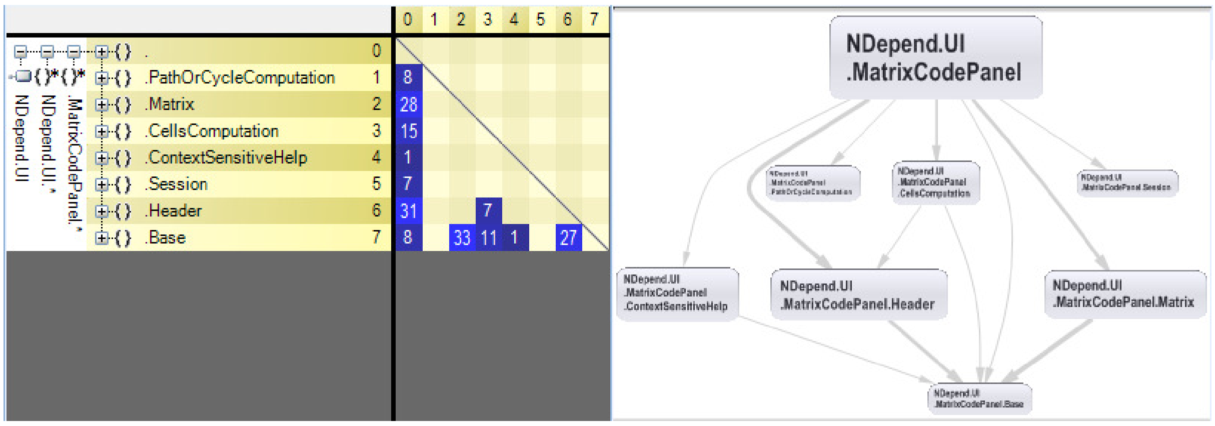
The size of a component must be reasonable, between 500 and 2000 LoC. Indeed, we suggested already that a component is a *unit of learning* and a *unit of architecture*. A component must never be too large to be reviewed and understood. An architecture that is made of such components becomes coarse, and leads to a monolithic code style and uncontrolled entropy.

The size of 500 to 2.000 LoC is just inferred guideline based on real-world observations. 500 to 2.000 LoC usually represents from one to two dozen classes. In the case of an abstract component, made of interfaces and enumerations, the number of LoC can be zero. You'll agree that LoC is not the appropriate metric for abstract component size.

Structuring larger components

If a larger component is required, then the component classes should be divided within sub-namespaces. Because namespaces can be represented through a hierarchy, we can use this as a convenient way to partition super-components into smaller components.

For example, the Dependency Structure Matrix (DSM) of NDepend is a complex feature that weighs around 5.000 LoC: Because of this, the DSM implementation resides in a namespace that contains seven sub-namespaces. The sub-namespaces correspond to sub-features of the DSM such as Header handling and Cells Computation. The screenshot below shows this organization through the DSM itself and through a graph of dependencies:



NDepend.UI
NDepend.UI.*
.MatrixCodePanel.*
NDepend.UI .MatrixCodePanel
NDepend.UI .MatrixCodePanel .PathOrCycleComputation
NDepend.UI .MatrixCodePanel .CellsComputation
NDepend.UI .MatrixCodePanel.Session
NDepend.UI .MatrixCodePanel .ContextSensitiveHelp
NDepend.UI .MatrixCodePanel.Header
NDepend.UI .MatrixCodePanel.Matrix
NDepend.UI .MatrixCodePanel.Base

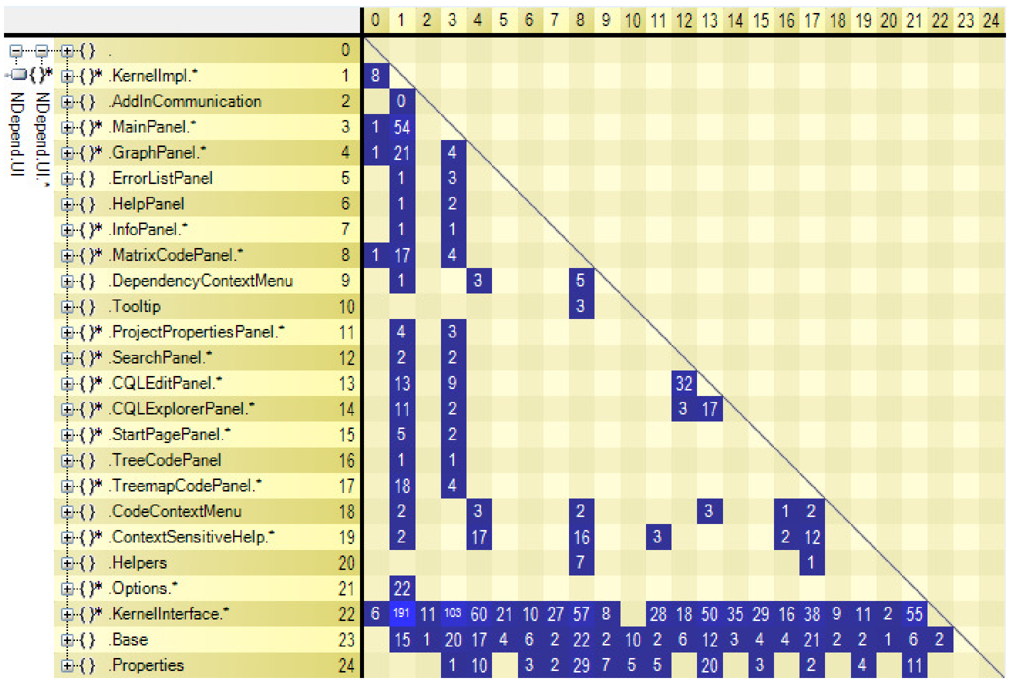
		0	1	2	3	4	5	6	7
.	0								
.PathOrCycleComputation	1	8							
.Matrix	2	28							
.CellsComputation	3	15							
.ContextSensitiveHelp	4	1							
.Session	5	7							
.Header	6	31			7				
.Base	7	8		33	11	1		27	

We can see that the graph is acyclic and that the parent namespace is using all child namespaces. This illustrates a recurring pattern where:

- The parent component plays the role of a mediator. Most of the sub-namespaces don't know about other sub-namespaces. This is done to reduce any chance of coupling. The mediator is responsible for making sub-components communicate.
- There is a base sub-component (the *Base* namespace) that is used by child sub-components and by the parent component. The base component mostly defines shared interfaces, enumerations and data classes that model the component domain. In our case it defines concepts such as matrix cell, matrix'-row/column header, dependency and dependency cycle in matrix or matrix display settings.
- It is not shown here, but the rest of the program only knows about the parent component. Hence using the namespace hierarchy is also a way to encapsulate some component implementation details. Here the parent namespace has two responsibilities: it defines the public surface visible from the rest of the program and it implements the role of a mediator between sub-components. It could have been even cleaner to separate these two responsibilities in:
    - the parent namespace, that would define the component public surface.
    - a single sub-namespace named *Impl*, that contains the mediator code and that contains other sub-namespaces.

Structuring code with Mediator, Feature and Base components

A program code base is likely to be composed of large components, each one of which will probably contain sub-components that are only needed for their own private implementations. This idea is illustrated by the DSM below that shows the architecture of the NDepend UI code (> 50K LoC). There are many large components in the list those who namespace name ends up with *.** to signify that they contains sub-components.



		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
{ } .	0																									
{ }* .KernelImpl.*	1	8																								
{ } .AddInCommunication	2		0																							
{ }* .MainPanel.*	3	1	54																							
{ }* .GraphPanel.*	4	1	21		4																					
{ } .ErrorListPanel	5		1		3																					
{ } .HelpPanel	6		1		2																					
{ }* .InfoPanel.*	7		1		1																					
{ }* .MatrixCodePanel.*	8	1	17		4																					
{ } .DependencyContextMenu	9		1			3				5																
{ } .Tooltip	10									3																
{ }* .ProjectPropertiesPanel.*	11		4		3																					
{ }* .SearchPanel.*	12		2		2																					
{ }* .CQLEditPanel.*	13		13		9									32												
{ }* .CQLExplorerPanel.*	14		11		2									3	17											
{ }* .StartPagePanel.*	15		5		2																					
{ } .TreeCodePanel	16		1		1																					
{ }* .TreemapCodePanel.*	17		18		4																					
{ } .CodeContextMenu	18		2			3				2					3			1	2							
{ }* .ContextSensitiveHelp.*	19		2			17				16			3					2	12							
{ } .Helpers	20									7									1							
{ }* .Options.*	21		22																							
{ }* .KernelInterface.*	22	6	191	11	103	60	21	10	27	57	8		28	18	50	35	29	16	38	9	11	2	55			
{ } .Base	23		15	1	20	17	4	6	2	22	2	10	2	6	12	3	4	4	21	2	2	1	6	2		
{ } .Properties	24				1	10		3	2	29	7	5	5		20		3		2		4		11			

The two vertical lines on *KernelImpl.**and *MainPanel.** namespaces (columns 1 and 3) indicate that these two components act as mediators between lower level components. Hence the other columns are pretty empty, meaning that components that represent features (Matrix, Graph, CQL …) are independent from each other.

The three rows 22, 23 and 24, are almost full. They represent base components that support all higher level components. Obviously this high-level architecture and the finer-grained architecture of the Matrix components are pretty similar. The same pattern is applied at two different scales. In both cases there are:

- **Feature Components**: Independent components that contain features implementation.
- **Mediator Components**: A few high-level components that act as mediator between feature components. The mediator contains the plumbing needed to make features communicate between each others.
- **Base Components**: A few low-level components that implement the domain of the application. Base components classes are shared amongst feature components.

Main benefits of classifying components between feature, mediator and base are that:

- The clarity of the overall architecture is not blurred by the implementation details.
- The developer can zoom in on the program structure to understand it.
- The structure favors a low incidence of coupling between implementations details. (this is the Low-Coupling)
- The implementation of a particular feature is thoroughly nested in a well-identified root namespace (the is the High-Cohesion Pattern)

Acyclic graph of dependencies between components

By looking back to the DSM representing the NDepend.UI structure, we can see that the upper triangle above the matrix's diagonal is empty. Each component can be used by components above it, and can use components below it. Hence the structure is perfectly layered. It is important to remember that:

The graph of dependencies between components is acyclic if, and only if, its DSM representation has its upper triangle empty.

Since components are layered, each component can have a level index. Hence, we say that when an architecture has no dependency cycles, it is **levelized**. We have already tried to explain that a levelized architecture is essential because:

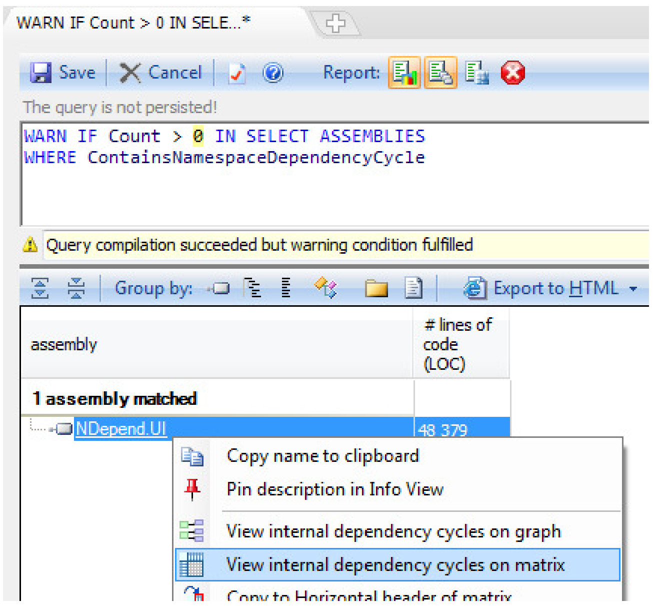
'If the graph of dependencies between components contains a cycle, components involved in the cycle cannot be developed and tested independently. Because of this, the cycle of components represents a super-component, with higher entropy than the sum of the entropies of its contained components.'

Every developer has the instinct to lean toward an acyclic structure. This natural tendency explains the popularity of Visual Studio (VS) solutions with dozens of small VS projects. VS detects, and prevents, dependency cycles between VS projects; therefore developers see in the VS project the ideal device with which to implement the idea of levelized component. Unfortunately, doing so is far from ideal and we explain in the article **Partitioning code base through .NET assemblies and Visual Studio Projects** all the real-world problems of this approach.

The tool NDepend offers a simple way of checking for dependency cycles between namespaces of an assembly. The following rule, written with Code Query Language (CQL), is all what one needs to be advised of a broken architecture.

```
WARN IF Count > 0 IN SELECT ASSEMBLIES WHERE ContainsNamespaceDependencyCycle
```

If an assembly is matched, just right-click the assembly in the list of match and click :*View internal dependency cycles on matrix* :



```
WARN IF Count > 0 IN SELECT ASSEMBLIES
WHERE ContainsNamespaceDependencyCycle
```

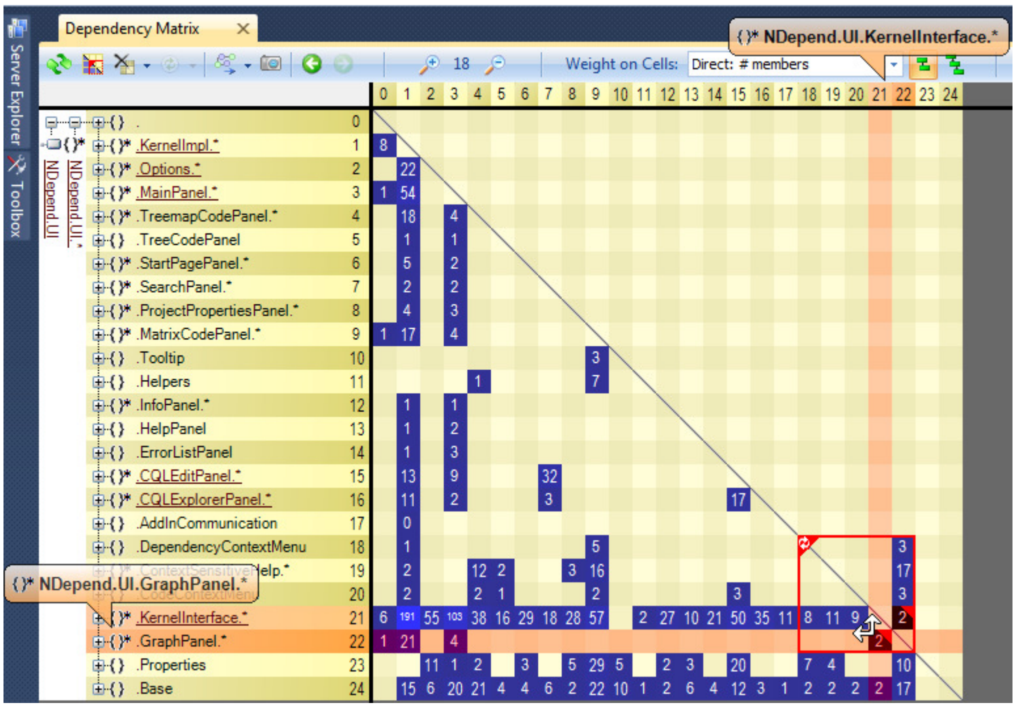
The query is not persisted!

Query compilation succeeded but warning condition fulfilled

assembly	# lines of code (LOC)
1 assembly matched	
NDepend.UI	48 379

WARN IF Count > 0 IN SELE...*
Save
Cancel
Report:
Group by:
Export to HTML
Copy name to clipboard
Pin description in Info View
View internal dependency cycles on graph
View internal dependency cycles on matrix
Copy to Horizontal header of matrix

The matrix will make the dependency cycle obvious with a red-square. Here, for the requirements of this article, we contrived it so that the two namepaces *GraphPanel* and *KernelInterface* are mutually dependent. This provokes five components to be entangled in a dependency cycle, hence the red square on the DSM that encompasses these five namespaces:



Weight on Cells: Direct: # members

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
{} .	0																									
{}* .KernelImpl.*	1	8																								
{}* .Options.*	2		22																							
{}* .MainPanel.*	3	1	54																							
{}* .TreemapCodePanel.*	4		18		4																					
{} .TreeCodePanel	5		1		1																					
{}* .StartPagePanel.*	6		5		2																					
{}* .SearchPanel.*	7		2		2																					
{}* .ProjectPropertiesPanel.*	8		4		3																					
{}* .MatrixCodePanel.*	9	1	17		4																					
{} .Tooltip	10										3															
{} .Helpers	11					1					7															
{}* .InfoPanel.*	12		1		1																					
{} .HelpPanel	13		1		2																					
{} .ErrorListPanel	14		1		3																					
{}* .CQLEditPanel.*	15		13		9				32																	
{}* .CQLExplorerPanel.*	16		11		2				3								17									
{} .AddInCommunication	17		0																							
{} .DependencyContextMenu	18		1								5													3		
{}* .ContextSensitiveHelp.*	19		2			12	2			3	16													17		
{} .CodeContextMenu	20		2			2	1				2						3							3		
{}* .KernelInterface.*	21	6	191	55	103	38	16	29	18	28	57		2	27	10	21	50	35	11	8	11	9		2		
{}* .GraphPanel.*	22	1	21		4																		2			
{} .Properties	23			11	1	2		3		5	29	5		2	3		20			7	4			10		
{} .Base	24		15	6	20	21	4	4	6	2	22	10	1	2	6	4	12	3	1	2	2	2	2	17		

'Levelizing' existing code is often cheaper than expected

Having a levelized structure between namespaces is enough to keep the architecture clean and maintainable. We've often noticed in code that the code structure naturally tends towards being levelized. This is because of the developers instinct for the notion of layers, that we already mentioned.

In the DSM above, we see the namespace *Base* at the lower level in the architecture. For most developer, it would seem unnatural and awkward to create a dependency from *Base* to any other component. Base's concepts are not supposed to use anything, they are here to be used from other components.

Often, the code structure is naturally close to levelized but not thoroughly levelized. Tooling is needed to prevent these few wrong dependencies (like from *Base* to something else) that appears with time. Usually there are not so many of these dependencies to fix. Because of this, the rule more than the exception is that **often 'Levelizing' an existing code base is a cheap process that can be achieved in a few days of work**.

Evolutionary Design and Acyclic componentization

If the code structure is kept levelized, low-level components never get a chance to bubble-up in the architecture not because someone decided so, but because above components won't let it bubble-up. Like in traditional building architecture, the structure itself put the pressure on low level components. As long as the acyclic components constraint is continuously respected, the code base remains highly learnable and maintainable.

- In traditional building architecture, the gravity strength put the pressure on low level artifacts. This makes them more stable: 'stable' in the sense they are hard to move.
- In software architecture, abiding by the acyclic component idea put the pressure on low level components. This makes them more stable, in the sense that it is painful to refactor them. Empirically abstractions are less often subject to refactoring than implementations. it is, for this reason, a good idea that low-level components contain mostly abstractions (interfaces and enumerations) to avoid painful refactoring.

The beauty of levelized architecture is that it discards the need for most design decisions. This lack of up-front design is known as **evolutionary design**. Let's quote Martin Fowler on evolutionary design:

With evolutionary design, you expect the design to evolve slowly over the course of the programming exercise. There's no design at the beginning. You begin by coding a small amount of functionality, adding more functionality, and letting the design shift and shape.

With levelized evolutionary design**,** good design is implicitly and continuously maintained. There are no questions about what to do to implement a new requirement. When planning new code to implement a requirement that is unpredicted, one just has to consider its fan-in and fan-out (who will use this new code and who this new code will use). From this information and from the need to preserve levelization, one can infer the level and the right location where this new code will fit well. Sometime the need to introduce abstractions through a pattern like *injection of code* or *inversion of dependency* will arise, but only to preserve levelization, not because it seems cool to do so or the new fashion pattern book advises it. And releases after releases, iterations after iterations, the design will evolve seamlessly toward something continuously flawless and unpredictable. Like in traditional building architecture, the structure won't collapse.

Guidelines

- Use the concept of namespace to define boundaries of components.
- A namespace typically contains from one to two dozens of types, and has a reasonable size that fits in the 500 to 2.000 LoC range.
- Take the time to levelize your code-base components, it is certainly a cheaper task than expected, so the Return On Investment will be high.
- Continuously check that the components' dependency graph inside an assembly is acyclic.
- If a component is too large (> 2.000 LoC), then use sub-namespaces to divide it into a smaller set of related components.
- At any scale, classify components between high-level mediators, middle-level independent features, low-level base/domains.
- Having a 'levelized' set of components removes the need for most design decisions.